String Abstractions for String Verification *

Fang Yu[1], Tevfik Bultan[2], and Ben Hardekopf[2]

[1] National Chengchi University, Taipei, Taiwan
yuf@nccu.edu.tw
[2] University of California, Santa Barbara, CA, USA
{bultan, benh}@cs.ucsb.edu

Abstract. Verifying string manipulating programs is a crucial problem in computer security. String operations are used extensively within web applications to manipulate user input, and their erroneous use is the most common cause of security vulnerabilities in web applications. Unfortunately, verifying string manipulating programs is an undecidable problem in general and any approximate string analysis technique has an inherent tension between efficiency and precision. In this paper we present a set of sound abstractions for strings and string operations that allow for both efficient and precise verification of string manipulating programs. Particularly, we are able to verify properties that involve implicit relations among string variables. We first describe an abstraction called regular abstraction which enables us to perform string analysis using multi-track automata as a symbolic representation. We then introduce two other abstractions—alphabet abstraction and relation abstraction—that can be used in combination to tune the analysis precision and efficiency. We show that these abstractions form an abstraction lattice that generalizes the string analysis techniques studied previously in isolation, such as size analysis or non-relational string analysis. Finally, we empirically evaluate the effectiveness of these abstraction techniques with respect to several benchmarks and an open source application, demonstrating that our techniques can improve the performance without loss of accuracy of the analysis when a suitable abstraction class is selected.

1 Introduction

String manipulation errors are a leading cause of security vulnerabilities. For example, the top three vulnerabilities in the Open Web Application Security Project (OWASP)'s top ten list [12] are Cross-Site Scripting, Injection Flaws (e.g., SQL injection), and Malicious File Execution (MFE), all of which occur due to the erroneous sanitization and manipulation of input strings. This paper focuses on the problem of verifying assertions about a program's string-related properties, e.g., that a string variable at a given program point cannot contain a specific set of characters, or that a string variable must necessarily be a prefix of some other string variable. These types of assertions can be used to prove the absence of the vulnerabilities listed above, among others.

The string verification problem is undecidable even for simple string manipulating programs [20]. In this paper we present sound abstraction techniques that enable

* This work is supported by an NSC grant 99-2218-E-004-002-MY3 and NSF grants CCF-0916112 and CCF-0716095.

us to verify properties of string manipulating programs. We first present the *regular abstraction* in which values of string variables are represented as multi-track deterministic finite automata. Multi-track automata read tuples of characters as input instead of only single characters. Each string variable corresponds to a particular track (i.e., a particular position in the tuple). This representation enables relational string analysis [20], meaning that it can be used to verify assertions that depend on relationships among the string variables.

Although we use a symbolic automata representation in order to improve the scalability of our analysis, the size of a multi-track automaton representing a given string relation can be exponentially larger than that of traditional single-track automata representing the (less-precise) non-relational projections of that same relation. For example, the 2-track automaton for the relation $X_1 = cX_2$ (where X_1 and X_2 are string variables) is exponential in the length of c, whereas the projection to the single-track automaton for X_2 (which is Σ^*) is constant-size and the projection for X_1 (which is $c\Sigma^*$) is linear in the length of c. Moreover, the size of the alphabet for multi-track automata increases exponentially with the number of program variables.

In order to improve the scalability of our approach, we propose two additional string abstractions: 1) *Relation abstraction* selects sets of string variables to analyze relationally and analyzes the remaining string variables independently. The intent is to detect relationships only for the string variables whose relations are relevant for the assertions being verified. 2) *Alphabet abstraction* selects a subset of alphabet characters to analyze distinctly and merges the remaining alphabet characters into a special symbol. The intent is to track only those characters that are relevant for the assertions being verified.

These two abstractions are parameterized by the choices of which program variables are analyzed in relation to each other versus being analyzed separately and which characters are merged versus being kept distinct; thus, they actually form a family of string abstractions. We show that this family forms an abstraction lattice that generalizes previously existing string analyses into a single, cohesive framework that can be tuned to provide various trade-offs between precision and performance. In addition, we propose a set of heuristics for choosing useful points in this abstraction lattice based on the program being analyzed. Finally, we empirically show that these abstractions meet our goal of increasing the scalability of the relational string analysis while retaining the precision required to verify realistic programs.

2 A Motivating Example

In this section we give a motivating example that shows the need for relational string analysis based on the regular abstraction as well as the usefulness of the relation and alphabet abstractions proposed in this paper. Consider the simple PHP script shown in Figure 1. The script starts with assigning the user account and password provided in `_GET` array to two string variables `$usr` and `$passwd` in line 2 and line 3, respectively. Then, it assigns the result of the concatenation of `$usr` and `$passwd` to another string variable `$key` in line 4. The `echo` statement in line 6 is a sink statement. Since it uses data directly from a user input, a taint analysis will identify this as a Cross Site Scripting (XSS) vulnerability. That is, a malicious user may provide an input that contains the

```
1 <?php
2    $usr = $_GET["usr"];
3    $passwd = $_GET["passwd"];
4    $key = $usr . $passwd;
5    if($key == "admin1234")
6       echo "You are login as " . $usr ;
7 ?>
```

Fig. 1. A Simple Example

string constant `<script` and execute a command leading to an XSS attack. However, the assignment in line 4 states that `$usr` is the prefix of `$key`. The branch condition in line 5 enforces the value of `$key` to be `admin1234` and the value of `$usr` to be a prefix of `admin1234`. This ensures that the `echo` statement in line 6 cannot take any string that contains the string constant `<script` as its input. Hence, this simple script is not vulnerable and the taint analysis raises a false alarm.

Non-relational string analyses techniques (e.g., [5, 19]) will also raise a false alarm for this simple script. Such an analysis will first observe that in lines 1 and 2 variables `$usr` and `$passwd` can have any string value since they are assigned user input, and then conclude that in line 3, `$key` can be any string. Note that the relationship among `$key`, `$usr` and `$passwd` can *not* be tracked in non-relational string analysis. Although such an analysis can conclude that the value of `$key` is `admin1234` in line 6, it will still report that `$usr` can take any string value in line 6. Let the attack pattern for the XSS be specified as the following regular expression: Σ^* `<script` Σ^* where Σ denotes the alphabet which is the set of all ASCII symbols. A non-relational string analysis will conclude that the `echo` statement in line 6 is vulnerable, since the `echo` statement can take a string value that matches the attack pattern if `$usr` can take any string value in line 6.

In our string analysis we first use a *regular abstraction* to represent possible values of string variables in a program using multi-track automata [20]. To represent values of n string variables, we use an n-track automaton with an alphabet of size $|\Sigma|^n$. Hence, as the number of variables increases, relational analysis becomes intractable. We propose two additional abstraction techniques to improve the scalability of the relational string analysis. We use the *relation abstraction* to reduce n, and the *alphabet abstraction* to reduce $|\Sigma|$, while preserving sufficient precision to prove the required assertions.

The relation abstraction chooses subsets of string variables to track relationally while tracking the remaining variables independently. For the example shown in Figure 1, a relational analysis that tracks `$usr` and `$key` together but `$passwd` independently is sufficient to prevent the false alarm at line 6. The two track automaton for `$usr` and `$key` is able to keep the prefix relation between `$usr` and `$key` at line 4. At line 6 while `$key` is restricted to `admin1234`, `$usr` is a strict prefix of `admin1234`, and the relational analysis is able to conclude that the `echo` statement cannot take a value that matches the attack pattern.

The alphabet abstraction keeps a subset of the alphabet symbols distinct and merges the remaining symbols to a single abstract symbol. For example, the program in Figure 1

can be analyzed more efficiently by keeping only the character $<$ distinct and merging all other ASCII symbols into an abstract symbol $\star$, thus shrinking the alphabet from 256 characters to 2. That is, we can use one bit (instead of eight bits) plus some reserved characters to encode each track of the multi-track automaton. Under this encoding, the analysis can conclude that at line 6, `$usr` is a string with length 9 that only contains $\star$, and the `echo` statement cannot take any string value that matches the attack pattern ($\star^* < \star\star\star\star\star(< |\star)^*$ in this case).

The relation and alphabet abstractions can be used only with regular abstraction, or composed together. In the example above, by combining relation abstraction and alphabet abstraction, we are able to decrease the alphabet size of the multi-track automaton, i.e., $|\Sigma|^n$, from 256^3 symbols to $2^2 = 4$ symbols, and still keep sufficient information to prevent the false alarm.

On the other hand, instead of tracing relations among variables, one may simply sanitize user inputs to prevent potential vulnerabilities, e.g., using the following statement at line 2:

```
$usr = str_replace("<", "", $_GET["usr"]);
```

The expression `_GET["usr"]` returns the string entered by the user, and the `str_replace` call replaces all $<$ with the empty string. In this case, we can adopt a coarser abstraction for our string analysis. It is sufficient to use single-track automata (abstract away all relations among variables) with abstract alphabet $\{<, \star\}$ to conclude that the `echo` statement cannot take any string value that matches the attack pattern.

3 String Abstractions

In this section we first present the regular abstraction which allows us to analyze string manipulating programs using multi-track automata as a symbolic representation. Then we show that the relation and alphabet abstractions can be composed with the regular abstraction (and with each other) to obtain a family of abstractions.

3.1 Regular Abstraction

The regular abstraction maps a set of string tuples to a set of string tuples accepted by a multi-track automaton. This enables us to use deterministic finite state automata (DFAs) as a symbolic representation during string analysis. A *multi-track automaton* (or multi-track DFA) is a DFA that transitions on tuples of characters rather than single characters. For a given alphabet Σ let $\Sigma_\lambda = \Sigma \cup \{\lambda\}$, where $\lambda \notin \Sigma$ is a special padding character. An n-track alphabet is defined as $\Sigma^n = \Sigma_\lambda \times \cdots \times \Sigma_\lambda$ (n times). A track corresponds to a particular position in the n-tuple. A multi-track DFA is *aligned* iff for all words w accepted by the DFA, $w \in \Sigma^*\lambda^*$ (i.e., all padding is at the end of the word). Using aligned multi-track automata gives us a representation that is closed under intersection and can be converted to a canonical form after determinization and minimization. In the following, multi-track DFAs are assumed to be aligned unless explicitly stated otherwise.

The statements of a string manipulating program can be represented as word equations. A *word equation* is an equality relation between two terms, each of which is a finite concatenation of string variables and string constants. Regular abstraction abstracts a given program by mapping the word-equations representing the program statements to multi-track DFA. Since word equations can not be precisely represented using multi-track automata, we use the results presented in [20] to construct a sound abstraction of the given program (i.e., in the abstracted program the set of values that a variable can take is a superset of the possible values that a variable can take in the concrete program). Note that, since branch conditions can contain negated terms, we need to be able to construct both an over- and an under-approximation of a given word equation. We construct multi-track automata that precisely represent word equations when possible, and either over- or under-approximate the word equations (as desired) otherwise.

We define a function CONSTRUCT(*exp*:word equation, *b*:bool) that takes a word equation as input and returns a corresponding multi-track DFA, if necessary either over-approximating (if $b = +$) or under-approximating (if $b = -$). We use the CONSTRUCT function to soundly approximate all word equations *and* their boolean combinations, including existentially-quantified word equations. The boolean operations conjunction, disjunction, and negation on word equations are handled using intersection, disjunction, and complementation of the corresponding multi-track DFAs, respectively; existentially-quantified word equations are handled using homomorphisms (by projecting the track that corresponds to the quantified variable).

Given an assignment statement $stmt$ of the form $X := exp$ we first represent it as a word equation of the form $X' = exp$ where exp is an expression on the current state variables, and X' denotes the next state variables. Then we abstract $stmt$ by constructing a multi-track automaton M_{stmt} that over-approximates the corresponding word equation as follows $M_{stmt} = \text{CONSTRUCT}(X' = exp, +)$. A branch condition specified as an expression exp is similarly abstracted using $\text{CONSTRUCT}(X' = X \wedge exp, +)$ for the then branch and $\text{CONSTRUCT}(X' = X \wedge \neg exp, +)$ for the else branch. The result of the regular abstraction consists of the control flow graph of the original program where each statement in the control flow graph is associated with a multi-track DFA that over-approximates the behavior of the corresponding statement.

The abstract domain that results from the regular abstraction is defined as a lattice on multi-track automata over an alphabet Σ^n. We denote this automata lattice as $\mathcal{L}_M = (\overline{M_{\Sigma^n}}, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$, where $\overline{M_{\Sigma^n}}$ is the set of multi-track automata over the alphabet Σ^n. For $M_1, M_2 \in \overline{M_{\Sigma^n}}$, $M_1 \sqsubseteq M_2$ iff $L(M_1) \subseteq L(M_2)$. The bottom element is defined as $L(\bot) = \emptyset$ and the top element is defined as $L(\top) = (\Sigma^n)^*$. There may be multiple automata that accept the same language; the lattice treats these automata as equivalent. If we use minimized DFAs then there is a unique automaton for each point in the lattice up to isomorphism. All of the multi-track automata in this lattice are aligned [20] and hence all operations take aligned automata as input and return aligned automata as output.

The join operator cannot be defined simply as language union since the family of regular languages is not closed under infinite union. Instead, we use the widening operator from [2] as the join operator where $M_1 \sqcup M_2 = M_1 \nabla M_2$. The meet operator

can be defined from the join operator using language complement: let $\neg M$ denote an automaton such that $L(\neg M) = \Sigma^* \setminus L(M)$; then $M_1 \sqcap M_2 = \neg(\neg M_1 \nabla \neg M_2)$.

Note that a similar automata lattice can also be defined for single-track automata over a single-track alphabet Σ where $\mathcal{L}_M = (\overline{M_\Sigma}, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$.

Fixpoint Computation for Forward Reachability The relational string analysis corresponds to a least-fixpoint computation over the multi-track automata lattice. Each program point is associated with a multi-track DFA whose tracks correspond to string variables and the multi-track DFA accepts the string-tuples that correspond to possible values that string variables can take at that particular program point. In order to be able to handle large alphabets of the multi-track DFA, we use the symbolic DFA representation provided by the MONA automata package [9]. In this symbolic automata representation the transition relations of the DFA are represented as Multi-terminal Binary Decision Diagrams (MBDDs).

We use a standard work-queue algorithm to compute the fixpoint. For a program statement *stmt* that is abstracted as the multi-track automaton M_{stmt} the post-image is computed as:

$$\textsc{post}(M, stmt) \equiv (\exists X.M \cap M_{stmt})[X' \mapsto X]$$

In other words, we take the intersection of the multi-track DFA that represents the statement with the multi-track DFA representing the current states (M), apply quantifier elimination on the current state variables X (by projection), and rename the next state variables X' as X (by arranging the indices in the MBDD representation) to obtain the post-image DFA.

Since we use the widening operator as the join operator during the fixpoint computation, the analysis is guaranteed to terminate. During the analysis, we report the assertion violations as they are discovered. The analysis is sound, but it is incomplete due to the following approximations: (1) abstraction of word equations as multi-track DFAs, and (2) use of the widening operator which over approximates the language union.

3.2 Alphabet Abstraction

In this section we formally define the alphabet abstraction. This abstraction targets the values taken on by string variables, mapping multiple alphabet symbols to a single abstract symbol. For example, consider a string variable that can take the value $\{ab, abc\}$. Abstracting the symbols b and c yields the value $\{a\star, a\star\star\}$, where $\star$ stands for both b and c. The concretization of this abstract value would yield the value $\{ab, ac, abc, abb, acb, acc\}$. At the extreme this abstraction can abstract out all alphabets symbols (in the above example, this would yield the abstract value $\{\star\star, \star\star\star\}$). In this case the only information retained from the original value is the length of the strings; all information about the content of the strings is lost. This abstraction is still useful in checking properties related to string length—we will return to this point in Section 3.4.

The alphabet abstraction is parameterized by the choice of which symbols to abstract, hence it forms a family of abstractions. This family forms an abstraction lattice $\mathcal{L}_\Sigma$ called the *alphabet lattice* (distinct from the automata lattice introduced earlier). Let Σ, a finite alphabet, be the concrete alphabet, and $\star \notin \Sigma$ be a special symbol

to represent characters that are abstracted away. An abstract alphabet of Σ is defined as $\Sigma' \cup \{\star\}$, where $\Sigma' \subseteq \Sigma$. The abstract alphabets of Σ form a complete lattice $\mathcal{L}_\Sigma = (\mathcal{P}(\Sigma \cup \{\star\}), \sqsubseteq_\Sigma, \cup, \cap, \sigma_\bot, \sigma_\top)$ where the bottom element $\sigma_\bot$ is $\Sigma \cup \{\star\}$, the top element $\sigma_\top$ is $\{\star\}$, and the join and meet operations correspond to set intersection and union, respectively. The abstraction $\sigma_\top$ corresponds to mapping all the symbols in the concrete alphabet to a single symbol, whereas $\sigma_\bot$ corresponds to no abstraction at all. The partial order of $\mathcal{L}_\Sigma$ is defined as follows. Let σ_1, σ_2 be two elements in $\mathcal{L}_\Sigma$,

$$\sigma_1 \sqsubseteq_\Sigma \sigma_2, \text{ if } \sigma_2 \subseteq \sigma_1, \quad \text{and} \quad \sigma_1 \sqsubset_\Sigma \sigma_2, \text{ if } \sigma_1 \sqsubseteq_\Sigma \sigma_2 \text{ and } \sigma_1 \neq \sigma_2.$$

Let $\sigma_1 \sqsubseteq_\Sigma \sigma_2$. We define the representation function for alphabet abstraction as follows: $\beta_{\sigma_1,\sigma_2} : \Sigma^* \to \Sigma^*$ where $\beta_{\sigma_1,\sigma_2}(w) = \{w' \mid |w'| = |w|, \forall i\ 1 \leq i \leq |w|.(w(i) \in \sigma_2 \Rightarrow w'(i) = w(i)) \wedge (w(i) \notin \sigma_2 \Rightarrow w'(i) = \star)\}$. The representation function simply maps the symbols that we wish to abstract to the abstract symbol $\star$, and maps the rest of the symbols to themselves.

Since the symbolic analysis we defined in Section 3.1 uses automata as a symbolic representation, we have to determine how to apply the alphabet abstraction to automata. We define the abstraction function $\alpha_{\sigma_1,\sigma_2}$ on automata using the representation function $\beta_{\sigma_1,\sigma_2}$ as follows: Let M be a single track DFA over σ_1; then $\alpha_{\sigma_1,\sigma_2}(M) = M'$ where M' is a single track DFA over σ_2 such that $L(M') = \{w \mid \exists w' \in L(M).\beta_{\sigma_1,\sigma_2}(w') = w\}$. Note that there may be multiple automata M' that satisfies this constraint. However, since we use minimized multi-track DFAs they will all be equivalent. We define the concretization function $\gamma_{\sigma_2,\sigma_1}$ similarly: Let M be a single track DFA over σ_2; then $\gamma_{\sigma_1,\sigma_2}(M) = M'$ where M' is a single track DFA over σ_1 such that $L(M') = \{w \mid \exists w' \in L(M).\beta_{\sigma_1,\sigma_2}(w) = w'\}$.

The definitions we give above are not constructive. We give a constructive definition of the abstraction and concretization functions by first defining an alphabet-abstraction-transducer that maps symbols that we wish to abstract to the abstract symbol $\star$, and maps the rest of the symbols to themselves.

An alphabet-abstraction-transducer over σ_1 and σ_2 is a 2-track DFA $M_{\sigma_1,\sigma_2} = \langle Q, \sigma_1 \times \sigma_2, \delta, q_0, F\rangle$, where

- $Q = \{q_0, sink\}$, $F = \{q_0\}$, and
- $\forall a \in \sigma_2.\delta(q_0, (a, a)) = q_0$,
- $\forall a \in \sigma_1 \setminus \sigma_2.\delta(q_0, (a, \star)) = q_0$.

Now, using the alphabet-abstraction-transducer, we can compute the abstraction of a DFA as a post-image computation, and we can compute the concretization of DFA as a pre-image computation. Let M be a single track DFA over σ_1 with track X. $M_{\sigma_1,\sigma_2}(X, X')$ denotes the alphabet transducer over σ_1 and σ_2 where X and X' correspond to the input and output tracks, respectively. We define the abstraction and concretization functions on automata as:

- $\alpha_{\sigma_1,\sigma_2}(M) \equiv (\exists X.M \cap M_{\sigma_1,\sigma_2}(X, X'))[X' \mapsto X]$, and
- $\gamma_{\sigma_1,\sigma_2}(M) \equiv \exists X'.(M[X \mapsto X'] \cap M_{\sigma_1,\sigma_2}(X, X'))$.

The definition can be extended to multi-track DFAs. Let M be a multi-track DFA over σ_1^n associated with $\{X_i \mid 1 \leq i \leq n\}$, $\alpha_{\sigma_1^n,\sigma_2^n}(M)$ returns a multi-track DFA over σ_2^n. On the other hand, while M is a multi-track DFA over σ_2^n, $\gamma_{\sigma_1^n,\sigma_2^n}(M)$ returns a multi-track DFA over σ_1^n. We use $M_{\sigma_1^n,\sigma_2^n}$ to denote the extension of the alphabet transducer to multi-track alphabet, where we add $\delta(q_0,(\lambda,\lambda)) = q_0$ to $M_{\sigma_1^n,\sigma_2^n}$ to deal with the padding symbol λ and we use $M_{\sigma_1^n,\sigma_2^n}(X_i, X_i')$ to denote the alphabet transducer associated with tracks X_i and X_i'. The abstraction and concretization of a multi-track DFA M is done track by track as follows:

- $\alpha_{\sigma_1^n,\sigma_2^n}(M) \equiv \forall X_i.(\exists X_i.M \cap M_{\sigma_1^n,\sigma_2^n}(X_i, X_i'))[X_i' \mapsto X_i]$, and
- $\gamma_{\sigma_1^n,\sigma_2^n}(M) \equiv \forall X_i.(\exists X_i'.M[X_i \mapsto X_i'] \cap M_{\sigma_1^n,\sigma_2^n}(X_i, X_i'))$.

The abstraction lattice $\mathcal{L}_\Sigma$ defines a family of Galois connections between the automata lattices $\mathcal{L}_M$. Each element σ^n in the abstraction lattice $\mathcal{L}_{\Sigma^n}$ is associated with an automata lattice $\mathcal{L}_{\sigma^n}$ corresponding to multi-track automata with the alphabet σ^n. For any pair of elements in the abstraction lattice $\sigma_1^n, \sigma_2^n \in \mathcal{L}_{\Sigma^n}$, if $\sigma_1^n \sqsubset_\Sigma \sigma_2^n$, then we can define a Galois connection between the corresponding automata lattices $\mathcal{L}_{\sigma_1^n}$ and $\mathcal{L}_{\sigma_2^n}$ using the abstraction and concretization functions $\alpha_{\sigma_1^n,\sigma_2^n}$ and $\gamma_{\sigma_1^n,\sigma_2^n}$. We formalize this with the following property:

Lemma 1. *For any Σ^n, and $\sigma_1^n, \sigma_2^n \in \mathcal{L}_{\Sigma^n}$, if $\sigma_1^n \sqsubset_\Sigma \sigma_2^n$, the functions $\alpha_{\sigma_1^n,\sigma_2^n}$ and $\gamma_{\sigma_1^n,\sigma_2^n}$ define a Galois connection between the lattices $\mathcal{L}_{\sigma_1^n}$ and $\mathcal{L}_{\sigma_2^n}$ where for any $M_1 \in \mathcal{L}_{\sigma_1^n}$ and $M_2 \in \mathcal{L}_{\sigma_2^n}$:*

$$\alpha_{\sigma_1^n,\sigma_2^n}(M_1) \sqsubseteq M_2 \;\Leftrightarrow\; M_1 \sqsubseteq \gamma_{\sigma_1^n,\sigma_2^n}(M_2)$$

3.3 Relation Abstraction

In this section we formally define the relation abstraction. This abstraction targets the relations between string variables. The abstraction determines the sets of variables that will be analyzed in relation to each other; for each such set the analysis computes a multi-track automaton for each program point such that each track of the automaton corresponds to one variable in that set. In the most abstract case no relations are tracked at all—there is a separate single-track automaton for each variable and the analysis is completely non-relational. On the other hand, in the most precise case we have one single multi-track automaton for each program point.

Let $\overline{X} = \{X_1, \ldots X_n\}$ be a finite set of variables. Let $\chi \subseteq 2^{\overline{X}}$ where $\emptyset \notin \chi$. We say χ defines a relation of $\overline{X}$ if (1) for any $\mathbf{x}, \mathbf{x}' \in \chi$, $\mathbf{x} \not\subseteq \mathbf{x}'$, and (2) $\bigcup_{\mathbf{x}\in\chi} \mathbf{x} = \overline{X}$. The set of χ that defines the relations of $\overline{X}$ form a complete lattice, denoted as $\mathcal{L}_{\overline{X}}$.

- The bottom of the abstraction lattice, denoted as $\chi_\perp$, is $\{\{X_1, X_2, \ldots, X_n\}\}$. This corresponds to the most precise case where, for each program point, a single multi-track automaton is used to represent the set of values for all string variables where each string variable corresponds to one track. This is the representation used in the symbolic reachability analysis described in Section 3.1.

– The top of the abstraction lattice, denoted as $\chi_\top$, is $\{\{X_1\}, \{X_2\}, \{X_3\}, \ldots, \{X_n\}\}$. This corresponds to the most coarse abstraction where, for each program point, n single-track automata are used and each automaton represents the set of values for a single string variable. This approach has been used in some earlier work such as [1, 19].

The partial order of the abstraction lattice $\mathcal{L}_{\overline{X}}$ is defined as follows: Let χ_1, χ_2 be two elements in $\mathcal{L}_{\overline{X}}$,

– $\chi_1 \sqsubseteq_{\overline{X}} \chi_2$, if for any $\mathbf{x} \in \chi_2$, there exists $\mathbf{x}' \in \chi_1$ such that $\mathbf{x} \subseteq \mathbf{x}'$.
– $\chi_1 \sqsubset_{\overline{X}} \chi_2$ if $\chi_1 \sqsubseteq_{\overline{X}} \chi_2$ and $\chi_1 \neq \chi_2$.

The symbolic reachability analysis discussed in Section 3.1 can be generalized to a symbolic reachability analysis that works for each abstraction level in the abstraction lattice $\mathcal{L}_{\overline{X}}$. To conduct symbolic reachability analysis for the relation abstraction $\chi \in \mathcal{L}_{\overline{X}}$, we store $|\chi|$ multi-track automata for each program point, where for each $\mathbf{x} \in \chi$, we have a $|\mathbf{x}|$-track DFA, denoted as $M_{\mathbf{x}}$, where each track is associated with a variable in $\mathbf{x}$.

In order to define the abstraction and the concretization functions, we define the following projection and extension operations on automata. For $\mathbf{x}' \subseteq \mathbf{x}$, the projection of $M_{\mathbf{x}}$ to $\mathbf{x}'$, denoted as $M_{\mathbf{x}} \downarrow_{\mathbf{x}'}$, is defined as the $|\mathbf{x}'|$-track DFA that accepts $\{w' \mid w \in L(M_{\mathbf{x}}), \forall X_i \in \mathbf{x}'.w'[i] = w[i])\}$. Similarly, $\mathbf{x}' \subseteq \mathbf{x}$, the extension of $M_{\mathbf{x}'}$ to $\mathbf{x}$, denoted as $M_{\mathbf{x}'} \uparrow_{\mathbf{x}}$, is defined as the $|\mathbf{x}|$-track DFA that accepts $\{w \mid w' \in L(M_{\mathbf{x}'}), \forall X_i \in \mathbf{x}'.w[i] = w'[i])\}$.

Let $\mathbf{M}_\chi = \{M_{\mathbf{x}} \mid \mathbf{x} \in \chi\}$ be a set of DFAs for the relation χ. The set of string values represented by $\mathbf{M}_\chi$ is defined as: $L(\mathbf{M}_\chi) = L(\bigcap_{\mathbf{x} \in \chi} M_{\mathbf{x}} \uparrow_{\mathbf{x_u}})$, where $\mathbf{x_u} = \{X_1, X_2, \ldots, X_n\}$. I.e., we extend the language of every automaton in $\mathbf{M}_\chi$ to all string variables and then take their intersection.

Now, let us define the abstraction and concretization functions for the relation abstraction (which take a set of multi-track automata as input and return a set of multi-track automata as output).

Let $\chi_1 \sqsubset_{\overline{X}} \chi_2$; then $\alpha_{\chi_1,\chi_2}(\mathbf{M}_{\chi_1})$ returns a set of DFAs $\{M_{\mathbf{x}'} \mid \mathbf{x}' \in \chi_2\}$, where for each $\mathbf{x}' \in \chi_2$, $M_{\mathbf{x}'} = (\bigcap_{\mathbf{x} \in \chi_1, \mathbf{x}' \cap \mathbf{x} \neq \emptyset} M_{\mathbf{x}} \uparrow_{\mathbf{x_u}}) \downarrow_{\mathbf{x}'}$, where $\mathbf{x_u} = \{X_i \mid X_i \in \mathbf{x}, \mathbf{x} \in \chi_1, \mathbf{x}' \cap \mathbf{x} \neq \emptyset\}$.

$\gamma_{\chi_1,\chi_2}(\mathbf{M}_{\chi_2})$ returns a set of DFAs $\{M_{\mathbf{x}} \mid \mathbf{x} \in \chi_1\}$, where for each $\mathbf{x} \in \chi_1$, $M_{\mathbf{x}} = (\bigcap_{\mathbf{x}' \in \chi_2, \mathbf{x}' \cap \mathbf{x} \neq \emptyset} (M_{\mathbf{x}'} \uparrow_{\mathbf{x_u}})) \downarrow_{\mathbf{x}}$, where $\mathbf{x_u} = \{X_i \mid X_i \in \mathbf{x}', \mathbf{x}' \in \chi_2, \mathbf{x}' \cap \mathbf{x} \neq \emptyset\}$.

Similar to the alphabet abstraction, the relation abstraction lattice $\mathcal{L}_{\overline{X}}$ also defines a family of Galois connections. Each element of the relation abstraction lattice corresponds to a lattice on sets of automata. For each $\chi \in \mathcal{L}_{\overline{X}}$ we define a lattice $\mathcal{L}_\chi = (\overline{\mathbf{M}_\chi}, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$. Given two sets of automata $\mathbf{M}_\chi, \mathbf{M}'_\chi \in \overline{\mathbf{M}_\chi}$, $\mathbf{M}_\chi \sqsubseteq \mathbf{M}'_\chi$ if and only if $L(\mathbf{M}_\chi) \subseteq L(\mathbf{M}'_\chi)$. The bottom element is defined as $L(\bot) = \emptyset$ and the top element is defined as $L(\top) = (\Sigma^n)^*$. The join operator is defined as: $\mathbf{M}_\chi \sqcup \mathbf{M}'_\chi = \{M_{\mathbf{x}} \nabla M'_{\mathbf{x}} \mid \mathbf{x} \in \chi, M_{\mathbf{x}} \in \mathbf{M}_\chi, M'_{\mathbf{x}} \in \mathbf{M}'_\chi\}$ and the meet operator is defined as: $\mathbf{M}_\chi \sqcap \mathbf{M}'_\chi = \{\neg(\neg M_{\mathbf{x}} \nabla \neg M'_{\mathbf{x}}) \mid \mathbf{x} \in \chi, M_{\mathbf{x}} \in \mathbf{M}_\chi, M'_{\mathbf{x}} \in \mathbf{M}'_\chi\}$.

For any pair of elements in the relation abstraction lattice $\chi_1, \chi_2 \in \mathcal{L}_{\overline{X}}$, if $\chi_1 \sqsubset_{\overline{X}} \chi_2$, then the abstraction and concretization functions α_{χ_1,χ_2} and γ_{χ_1,χ_2} define a Galois connection between $\mathcal{L}_{\chi_1}$ and $\mathcal{L}_{\chi_2}$. We formalize this with the following property:

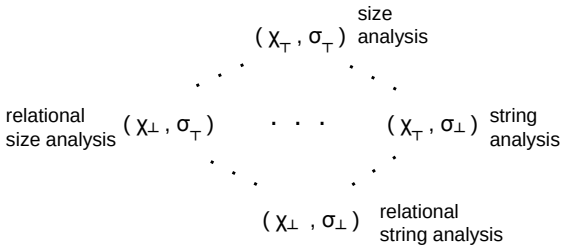

Fig. 2. Some abstractions from the abstraction lattice and corresponding analyses

Lemma 2. *For any $\chi_1, \chi_2 \in \mathcal{L}_{\overline{X}}$, if $\chi_1 \sqsubset_{\overline{X}} \chi_2$, then the functions α_{χ_1,χ_2} and γ_{χ_1,χ_2} define a Galois connection between $\mathcal{L}_{\chi_1}$ and $\mathcal{L}_{\chi_2}$ where for any $\mathbf{M}_{\chi_1} \in \mathcal{L}_{\chi_1}$ and $\mathbf{M}'_{\chi_2} \in \mathcal{L}_{\chi_2}$:*

$$\alpha_{\chi_1,\chi_2}(\mathbf{M}_{\chi_1}) \sqsubseteq \mathbf{M}'_{\chi_2} \;\Leftrightarrow\; \mathbf{M}_{\chi_1} \sqsubseteq \gamma_{\sigma_1^n,\sigma_2^n}(\mathbf{M}'_{\chi_2})$$

3.4 Composing Abstractions

As shown in the two previous sections, both alphabet and relation abstractions form abstraction lattices which allow different levels of abstraction. Combining these abstractions leads to a product lattice where each point in this lattice corresponds to the combination of a particular alphabet abstraction with a particular relation abstraction. This creates an even larger set of Galois connections, one for each possible combination of alphabet and relation abstractions. Given Σ and $\overline{X} = \{X_1, \ldots, X_n\}$, we define a point in this product lattice as an *abstraction class* which is a pair (χ, σ) where $\chi \in \mathcal{L}_{\overline{X}}$ and $\sigma \in \mathcal{L}_{\Sigma}$. The abstraction classes of $\overline{X}$ and Σ also form a complete lattice, of which the partial order is defined as: $(\chi_1, \sigma_1) \sqsubseteq (\chi_2, \sigma_2)$ if $\chi_1 \sqsubseteq \chi_2$ and $\sigma_1 \sqsubseteq \sigma_2$.

Given Σ and $\overline{X} = \{X_1, \ldots, X_n\}$, we can select any abstraction class in the product lattice during our analysis. The selected abstraction class (χ, σ) determines the precision and efficiency of our analysis. If we select the abstraction class $(\chi_\bot, \sigma_\bot)$, we conduct our most precise relational string analysis. The relations among $\overline{X}$ will be kept using one n-track DFA at each program point. If we select $(\chi_\top, \sigma_\top)$, we only keep track of the *length* of each string variable individually. Although we abstract away almost all string relations and contents in this case, this kind of path-sensitive (w.r.t length conditions on a single variable) size analysis can be used to detect buffer overflow vulnerabilities [6, 15]. If we select $(\chi_\bot, \sigma_\top)$, then we will be conducting relational size analysis. Finally, earlier string analysis techniques that use DFA, such as [1, 19], correspond to the abstraction class $(\chi_\top, \sigma_\bot)$, where multiple single-track DFAs over Σ are used to encode reachable states. As shown in [1, 17, 19], this type of analysis is useful for detecting XSS and SQLCI vulnerabilities.

Figure 2 summarizes the different types of abstractions that can be obtained using our abstraction framework. The alphabet and relation abstractions can be seen as two knobs that determine the level of the precision of our string analysis. The alphabet abstraction knob determines how much of the string content is abstracted away. In the

limit, the only information left about the string values is their lengths. On the other hand, the relation abstraction knob determines which set of variables should be analyzed in relation to each other. In the limit, all values are projected to individual variables. Different abstraction classes can be useful in different cases. An important question is how to choose a *proper* abstraction class for a given verification problem; this is addressed in the next section.

3.5 Heuristics and Refinement for Abstraction Selection

Since the space of total possible abstractions using the relation and alphabet abstractions defined above is very large, we must have some means to decide which specific abstractions to employ in order to prove a given property. Our choice should be as abstract as possible (for efficiency) while remaining precise enough to prove the property in question. In this section we propose a set of heuristics for making this decision.

The intuition behind these heuristics is to let the property under question guide our choice of abstraction. Consider the set of string variables X_i that appear in the assertion of a property. Let G be the dependence graph for X_i generated from the program being verified. Finally, let $\overline{X}$ be the set of string variables appearing in G and $\overline{C}$ be the set of characters used in any string constants that appear in G or in the assertion itself. It is the characters in $\overline{C}$ and the relations between the variables in $\overline{X}$ that are most relevant for proving the given property, and therefore these relations and characters should be preserved by the abstraction—the remaining variables and characters can be safely abstracted away without jeopardizing the verification.

In choosing the alphabet abstraction, our heuristic keeps distinct all characters appearing in $C \subseteq \overline{C}$ and merges all remaining characters of the alphabet together into the special character $\star$. Initially, C is the set of characters that appear in the assertion itself. C can be iteratively refined by adding selected characters in $\overline{C}$.

In choosing the relation abstraction, it is possible to simply group all variables in $\overline{X}$ together and track them relationally while tracking all other variables independently, but this choice might be more precise than necessary. Consider a hypothetical property $(X_1 = X_2 \wedge X_3 = X_4)$. It is not necessary to track the relations between X_1 and either of X_3 or X_4 (and similarly for X_2)—it is sufficient to track the relations between X_1 and X_2 and separately the relations between X_2 and X_3. Therefore our heuristic partitions $\overline{X}$ into distinct groups such that the variables within each group are tracked relationally with each other. To partition $\overline{X}$, our heuristic first groups together variables that appear in the same word equation in the property being verified (e.g., X_1 with X_2 and X_3 with X_4). The partition can be iteratively refined by using the dependency graph G to merge groups containing variables that depend on each other.

This heuristic can also be extended to take into account path sensitivity. Let ϕ be the path condition for the assertion (or the sink statement) and X_ϕ denote the set of string variables appearing in ϕ. We first extend the partition base from $\overline{X}$ to $\overline{X} \cup X_\phi$. Initially, each variable forms an individual group by itself. The partition is then refined by merging these individual groups with groups that contain variables that these new variables depend on. For instance, for the motivating example shown in Figure 1, we have $\overline{X} = \{\texttt{\$usr}\}$ and $X_\phi = \{\texttt{\$key}\}$. The initial partition is $\{\{\texttt{\$usr}\}, \{\texttt{\$key}\}\}$, and

the refined partition is $\{\{\texttt{\$usr}, \texttt{\$key}\}\}$ which enables `$usr` and `$key` to be tracked relationally.

3.6 Handling Complex String Operations

We extend our analysis to other complex string operations that have been previously defined using single-track automata [19, 21], e.g., replacement, prefix, and suffix. We first extract the set of values for each variables from the multi-track DFA $\mathbf{M}_\chi$ as a single track DFA (using projection), then compute the result of the string operation using these single-track DFAs. The post image of $\mathbf{M}_\chi$ can then be computed using the resulting DFAs.

We must modify these operations to ensure the soundness of the alphabet abstraction. Consider REPLACE(M_1, M_2, M_3) [19], which returns the DFA accepting $\{w_1c_1w_2c_2 \dots w_kc_kw_{k+1} \mid k > 0, w_1x_1w_2x_2 \dots w_kx_kw_{k+1} \in L(M_1), \forall_i, x_i \in L(M_2), w_i$ does not contain any substring accepted by $M_2, c_i \in L(M_3)\}$. As an instance, let $L(M_1) = \{ab\}$, $L(M_2) = \{c\}$, $L(M_3) = \{a\}$. REPLACE(M_1, M_2, M_3) will return M that accepts $\{ab\}$, the same as M_1, since there is no match appearing in any accepted string. However, let $\{a, \star\}$ be the abstraction alphabet. After applying the alphabet abstraction, we have $L(M_1') = \{a\star\}$, $L(M_2') = \{\star\}$, $L(M_3') = \{a\}$, and REPLACE(M_1', M_2', M_3') will return M' that accepts $\{aa\}$ instead. Since $L(\alpha_\sigma(M)) = \{a\star\} \not\subseteq L(M')$, the result in the concrete domain is not included in the abstract domain after abstraction. It is unsound applying the replace operation directly.

Assume M_1, M_2, M_3 using the same abstraction alphabet σ. To ensure soundness we return α_σ(REPLACE($\gamma_\sigma(M_1), \gamma_\sigma(M_2), \gamma_\sigma(M_3)$) if $L(M_1) \not\subseteq L(M_{\bar{\star}})$ and $L(M_2) \not\subseteq L(M_{\bar{\star}})$, so that all possible results in the concrete domain are included in the abstract domain after abstraction. We return REPLACE(M_1, M_2, M_3), otherwise.

4 Implementation and Experiments

We have incorporated alphabet abstraction to Stranger [18] which is an automata-based string analysis tool for PHP programs. However, Stranger at this point performs string analysis on dependency graphs and is limited to non-relational analysis, i.e., the bottom of the relation abstraction lattice. In order to implement our relational string analysis we extended the symbolic string analysis library that Stranger uses to support (abstract) relational analysis with multi-track automata. We compiled the extended Stranger string analysis library separate from the Stranger front-end to implement the relational string analysis. To evaluate the proposed relation abstraction, we implemented the relational analysis by directly calling the extended Stranger string analysis library functions.

We ran two experiments. In the first experiment we used two sets of benchmarks: 1) Malicious File Execution (MFE) benchmarks, and 2) Cross-Site Scripting (XSS) benchmarks. These benchmarks come from five open source PHP applications. We first used the Stranger front-end to conduct taint analysis on these PHP applications. Stranger taint-analyzer identified the tainted sinks (i.e., sensitive program points that might be vulnerable) for each type of vulnerability and generated the dependency graphs for these potentially vulnerable program segments. We then implemented the relational

string analysis for several of these potentially vulnerable program segments by using the extended Stranger string analysis library functions. We implemented the relational string analysis for several abstraction classes to evaluate the effectiveness of the proposed abstractions.

In the second experiment we evaluated the alphabet abstraction by directly using Stranger (that we extended with alphabet abstraction) on an open source web application, called Schoolmate. For this application we looked for XSS (Cross-Site Scripting) vulnerabilities and conducted the string analysis both with and without alphabet abstraction and compared the results. Both experiments were conducted on the Linux 2.6.35 machine equipped with Intel Pentium Dual CPU 2.80 GHz and memory 2.9GB. The applications used in our experiments are available at:
`http://soslab.nccu.edu.tw/applications`.

MFE benchmarks. This set of benchmarks demonstrates the usefulness of the relational analysis as well as the utility of our relation abstraction. We used 5 benchmarks:

M1: PBLguestbook-1.32, pblguestbook.php (536)	**M2**: MyEasyMarket-4.1, prod.php (94)
M3: MyEasyMarket-4.1, prod.php (189)	**M4**: php-fusion-6.01, db_backup.php (111)
M5: php-fusion-6.01, forums_prune.php (28)	

Each benchmark is extracted from an open source web application as described above and contains a program point that executes a file operation (include, fopen, etc) whose arguments may be influenced by external inputs. For example, **M1** corresponds to the program point at line 536 in pblguestbook.php distributed in the application PBLguestbook-1.32. Given an abstraction class and a benchmark, we implemented the corresponding string analysis using the extended Stranger string manipulation library. Our analysis constructs a multi-track DFA for each program point that over-approximates the set of strings that form the arguments. At the end of the analysis these DFAs are intersected with a multi-track DFA that characterizes the vulnerable strings that expose the program to MFE attacks. We report an error if the intersection is non-empty. None of the MFE benchmarks we analyzed contained an actual vulnerability. The MFE vulnerabilities correspond to scenarios such as a user accessing a file in another user's directory. Such vulnerabilities depend on the relation between two string values (for example, the user name and the directory name), hence an analysis that does not track the relations between string variables would raise false alarms.

XSS benchmarks. This set of benchmarks (again extracted from open source web applications) demonstrates the utility of the alphabet abstraction. We use 3 benchmarks:

S1: MyEasyMarket-4.1, trans.php (218)	**S2**: Aphpkb-0.71, saa.php(87)
S3: BloggIT 1.0, admin.php (23)	

To identify XSS attacks we use a predetermined set of attack patterns specified as a regular language. These attack patterns represent strings that potentially make a program vulnerable to XSS attacks. Again, given an abstraction class and a benchmark, we implemented the corresponding string analysis using the extended Stranger string manipulation library. Our analysis constructs multi-track DFAs to over-approximate the set of possible strings values at sinks and intersects these DFAs with the attack patterns

to detect potential vulnerabilities. All three benchmarks we analyzed were vulnerable. We modified the benchmarks to fix these vulnerabilities and create three new versions (S1', S2', and S3') that are secure against XSS attacks.

Experimental Results. The results for the MFE benchmarks are summarized in Table 1. All DFAs are symbolically encoded using MBDDs (where MBDDs are used to represent the transition relation of the DFA). The column labeled "state" shows the number of states of the DFA while the column labeled "bdd" shows the number of nodes in the MBDD that encodes the transition relation of the DFA (i.e., it corresponds to the size of the transition relation of the DFA). Note that the transition relation size decreases when we use a coarser alphabet abstraction. However, using a coarser alphabet may induce nondeterministic edges, and as shown in Table 1, in some cases, the number of states may increase after determinization and minimization.

The first set of columns use the abstraction $(\chi_\top, \sigma_\bot)$, i.e., a completely non-relational analysis using a full alphabet (similar to the analyses proposed in [18, 19]). This level of abstraction fails to prove the desired properties and raise false alarms, demonstrating the importance of a relational analysis. The next set of columns uses the abstraction $(\chi, \sigma_\bot)$, using our heuristic to track a subset of variables relationally. This level of abstraction is able to prove all of the desired properties, demonstrating the utility of both the relational analysis and of our heuristic. Finally, the last set of columns uses the abstraction (χ, σ), using our heuristics for both relation and alphabet abstraction. This level of abstraction is also able to verify all the desired properties, and does so with even better time and memory performance than the previous level of abstraction.

The results for the XSS benchmarks are summarized in Table 2. The first three rows show results for the unmodified benchmarks. Since our analysis is sound and all three benchmarks are vulnerable to XSS attacks, we cannot verify the desired properties regardless of the abstractions used. The last three rows show results for the modified benchmarks whose vulnerabilities have been patched, and therefore are secure. The first set of columns uses the abstraction $(\chi_\top, \sigma_\bot)$, i.e., a completely non-relational analysis using a full alphabet. This level of abstraction is sufficient to prove the desired properties, demonstrating that relational analysis is not always necessary and that the ability to selectively remove relational tracking is valuable. The last set of columns uses the abstraction $(\chi_\top, \sigma)$, using our alphabet abstraction to abstract some characters of the alphabet. This level of abstraction is also able to prove the desired properties and does so with improved time and memory performance than the previous level of abstraction, demonstrating again the benefit of our alphabet abstraction.

Detecting XSS Vulnerabilities in an Open-source Web Application. Our second experiment demonstrates the utility of the alphabet abstraction in detecting XSS vulnerabilities in an open source application: Schoolmate. Schoolmate consists of 63 php files with 8620 lines of code in total. The string analysis we report in this experiment is performed fully automatically using the Stranger tool that we extended with the alphabet abstraction.

The experimental results are summarized in Table 3. We first detect XSS vulnerabilities against the original code of schoolmate (denoted as O in Table 3). The first row shows the result of using $(\chi_\top, \sigma_\bot)$, a completely non-relational analysis using full set

	$(\chi_\top, \sigma_\bot)$				$(\chi, \sigma_\bot)$				(χ, σ)			
	Res.	DFA state(bdd)	Time (sec)	Mem (kb)	Res.	DFA state(bdd)	Time (sec)	Mem (kb)	Res.	DFA state(bdd)	Time (sec)	Mem (kb)
M1	n	56(801)	0.030	621	y	50(3551)	0.061	1294	y	54(556)	0.019	517
M2	n	22(495)	0.017	555	y	21(604)	0.044	996	y	22(179)	0.01	538
M3	n	5(113)	0.01	417	y	3(276)	0.019	465	y	3(49)	0.005	298
M4	n	1201(25949)	0.251	9495	y	181(9893)	0.854	19322	y	175(4137)	0.348	5945
M5	n	211(3195)	0.057	1676	y	62(2423)	0.102	1756	y	66(1173)	0.036	782

Table 1. Experimental results for the MFE benchmarks. DFA: the final DFA associated with the checked program point. state: number of states. bdd: number of BDD nodes. Result: "n" not verified, "y" verified.

	$(\chi_\top, \sigma_\bot)$				$(\chi, \sigma_\bot)$			
	Res.	DFA state(bdd)	Time (sec)	Mem (kb)	Res.	DFA state(bdd)	Time (sec)	Mem (kb)
S1	n	17(148)	0.012	444	n	65(1629)	0.345	1231
S2	n	27(229)	0.037	895	n	47(2714)	0.161	2684
S3	n	79(633)	0.067	1696	n	79(1900)	0.229	2826
	$(\chi_\top, \sigma_\bot)$				$(\chi_\top, \sigma)$			
S1'	y	17(147)	0.012	382	y	17(89)	0.006	287
S2'	y	17(141)	0.252	5686	y	9(48)	0.041	2155
S3'	y	127(1142)	0.444	6201	y	125(743)	0.299	3802

Table 2. Experimental results for the XSS benchmarks.

of ASCII characters as the alphabet. The second row shows the result of $(\chi_\top, \sigma)$, where we apply alphabet abstraction by keeping only the characters appearing in the attack pattern precisely. Each alphabet character is encoded using 3 bits. The coarser abstract analysis discovers 114 potential XSS vulnerabilities out of 898 sinks, using 1052 seconds for the string analysis (fwd) and 1104 seconds in total to explore all entries of 63 PHP scripts. The average size of a dependency graph (for a sink) has 33 nodes (each node represents one string operation) and 33 edges, while the maximum one has 123 nodes and 129 edges (the graph contains cycles). The average memory consumption is 62.5Mb for checking whether a sink is vulnerable, while the maximum consumption is 191Mb. The final DFA on average consists of 1051 states and 5255 bdd nodes to encode the transition relation, and the maximum one consists of 3593 states and 18005 bdd nodes. Compared to $(\chi_\top, \sigma_\bot)$, using alphabet abstraction, we introduce zero false alarms (number of reported vulnerabilities are 114 in both analyses) but reduce the analysis time 28% (from 1464 seconds to 1052) and reduce the memory usage 49% (from 62.5Mb to 31.9) on average.

Next we manually inserted 43 sanitization routines in the original code to remove the detected vulnerabilities by sanitizing user inputs and checked the resulting sanitized code against XSS vulnerabilities again (denoted as S in Table 3). The third row shows the result of using $(\chi_\top, \sigma_\bot)$, while the fourth row shows the result of $(\chi_\top, \sigma)$. For the sanitized code, using alphabet abstraction, we introduce zero false alarms (number of reported vulnerabilities are 10 in both analyses), but reduce the analysis time 34% (from 924 seconds to 609) and reduce the memory usage 47% (from 52.4Mb to 27.7) on average. We have observed that the 10 vulnerabilities that were reported after we inserted the sanitization routines are false positives due to some unmodeled built-in

functions (which are conservatively considered to return any possible string value) and path insensitive analysis. The 104 out of 114 vulnerabilities reported in the original version of the Schoolmate application are real vulnerabilities that are eliminated by the sanitization functions that we manually inserted to the application. To summarize, our experimental results demonstrate that using alphabet abstraction, we are able to considerably improve the performance without loss of accuracy of the analysis.

	Abstraction	Result	Time(s)	Mem (kb)	DFA: state/bdd		Dep. Graph: node/edge	
		#vuls/#sinks	fwd/total	avg/max	avg	max	avg	max
O	$(\chi_\top, \sigma_\bot)$	114/898	1464/1526	62568/191317	764/6894	2709/24382	33/33	123/129
O	$(\chi_\top, \sigma)$	114/898	1052/1104	31987/89488	1051/5255	3593/18005	33/33	123/129
S	$(\chi_\top, \sigma_\bot)$	10/898	924/979	52466/145901	725/6564	2164/19553	41/41	143/149
S	$(\chi_\top, \sigma)$	10/898	609/662	27774/82640	1136/5689	3466/17364	41/41	143/149

Table 3. Checking XSS Vulnerabilities in Schoolmate.

5 Related Work

Symbolic verification using automata have been investigated in other contexts (e.g., Bouajjani et al. [3, 4]). In this paper we focus specifically on verification of string manipulation operations, which is essential to detect and prevent web-related vulnerabilities.

String analysis has been widely studied due to its relevance for security. One influential approach has been *grammar-based* string analysis [5]. This approach uses a context-free grammar to represent the possible string operations and then over-approximates the resulting language by converting the grammar to a regular language. This form of analysis has been used to check for various types of errors in Web applications [8, 11, 16]. This analysis is not relational and cannot verify the simple programs we discussed in Section 2. Both Minamide [11] and Wassermann and Su [16] use multi-track DFAs, known as *transducers*, to model string replacement operations. There are also several recent string analysis tools that use symbolic string analysis based on DFA encodings [7, 14, 19, 21]. Some of these tools employ symbolic execution and use a DFA representation to model and verify string manipulation operations in Java [7, 14]. In our earlier work, we have used a DFA based symbolic reachability analysis to verify the correctness of string sanitization operations in PHP programs [19, 21].

Unlike the relational string analysis approach we use in this paper, (which is based on the results first presented by Yu et al. [20]) all of the above results use single-track DFA and encode the reachable configurations of each string variable separately—i.e., they use a non-relational string analysis. As demonstrated in this paper, a relational analysis enables verification of properties that cannot be verified with these earlier approaches.

However, relational string analysis can generate automata that are exponentially larger than the automata generated during non-relational string analysis. The alphabet and relation abstractions we present in this paper enable us to improve the performance of the relational string analysis by adjusting its precision. The earlier results on relational string analysis presented by Yu et al. [20] do not use any abstraction techniques.

While other work has employed abstraction techniques on automata [3], the novel abstractions we present in this paper are based on string values and relations among string variables. These abstractions allow useful heuristics based on the constants and relations appearing in the input program and the property.

Compared to string analysis techniques based on bounded string constraint solvers (e.g., HAMPI [10] and Kaluza [13]) an important differentiating characteristic of our approach is the fact that it is sound and can, therefore, be used to prove absence of string vulnerabilities.

Finally, this paper shows how string abstraction techniques that can be composed to form an abstraction lattice that subsumes the previous work on string analysis and size analysis. Our previous results, e.g., string analysis [19], composite (string+size) analysis [21], and relational string analysis [20] all become part of this abstraction lattice. This is the first such generalized string analysis result as far as we know.

6 Conclusions

As web applications are becoming more and more dominant, security vulnerabilities in them are becoming increasingly critical. The most common security vulnerabilities in web applications are due to improper sanitization of user inputs, which in turn are due to erroneous or improper use of string manipulation operations. In this paper we have focused on a relational string analysis that can be used to verify string manipulation operations in web applications. We presented two string abstraction techniques called alphabet and relation abstraction. These abstraction techniques enable us to adjust the precision and performance of our string analysis techniques. We also proposed a heuristic to statically determine the abstraction level and empirically demonstrated the effectiveness of our approach on open source web applications.

References

1. D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, C. Kruegel, E. Kirda, and G. Vigna. Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications. In *Proceedings of the Symposium on Security and Privacy*, 2008.
2. C. Bartzis and T. Bultan. Widening arithmetic automata. In *Proc. of the 16th International Conference on Computer Aided Verification*, pages 321–333, 2004.
3. A. Bouajjani, P. Habermehl, and T. Vojnar. Abstract regular model checking. In *Proc. of the 16th International Conference on Computer Aided Verification (CAV)*, pages 372–386, 2004.
4. A. Bouajjani, B. Jonsson, M. Nilsson, and T. Touili. Regular model checking. In *Proc. of the 12th International Conference on Computer Aided Verification*, pages 403–418, 2000.
5. A. S. Christensen, A. Møller, and M. I. Schwartzbach. Precise analysis of string expressions. In *Proc. of the 10th International Static Analysis Symposium (SAS)*, volume 2694 of *LNCS*, pages 1–18. Springer-Verlag, June 2003.
6. N. Dor, M. Rodeh, and M. Sagiv. CSSV: towards a realistic tool for statically detecting all buffer overflows in c. *SIGPLAN Not.*, 38(5):155–167, 2003.
7. X. Fu, X. Lu, B. Peltsverger, S. Chen, K. Qian, and L. Tao. A static analysis framework for detecting sql injection vulnerabilities. In *Proc. of the 31st Annual International Computer Software and Applications Conference - Vol. 1*, pages 87–96, Washington, DC, USA, 2007.

8. C. Gould, Z. Su, and P. Devanbu. Static checking of dynamically generated queries in database applications. In *Proc. of the 26th International Conference on Software Engineering*, pages 645–654, 2004.
9. J. G. Henriksen, J. L. Jensen, M. E. Jørgensen, N. Klarlund, R. Paige, T. Rauhe, and A. Sandholm. Mona: Monadic second-order logic in practice. In *Proc. of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 89–110, 1995.
10. Adam Kiezun, Vijay Ganesh, Philip J. Guo, Pieter Hooimeijer, and Michael D. Ernst. Hampi: a solver for string constraints. In *ISSTA*, pages 105–116, 2009.
11. Y. Minamide. Static approximation of dynamically generated web pages. In *Proc. of the 14th International World Wide Web Conference*, pages 432–441, 2005.
12. Open Web Application Security Project (OWASP). Top ten project. `http://www.owasp.org/`, May 2007.
13. Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. A symbolic execution framework for javascript. In *IEEE Symposium on Security and Privacy*, pages 513–528, 2010.
14. D. Shannon, S. Hajra, A. Lee, D. Zhan, and S. Khurshid. Abstracting symbolic execution with string analysis. In *TAICPART-MUTATION '07*, pages 13–22, DC, USA, 2007.
15. D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Proc. of the Network and Distributed System Security Symposium*, pages 3–17, 2000.
16. G. Wassermann and Z. Su. Sound and precise analysis of web applications for injection vulnerabilities. In *Proc. of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation*, pages 32–41, 2007.
17. F. Yu, M. Alkhalaf, and T. Bultan. Generating vulnerability signatures for string manipulating programs using automata-based forward and backward symbolic analyses. In *Proc. of the 24th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2009.
18. F. Yu, M. Alkhalaf, and T. Bultan. Stranger: An automata-based string analysis tool for php. In *Proc. of the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2010.
19. F. Yu, T. Bultan, M. Cova, and O. H. Ibarra. Symbolic string verification: An automata-based approach. In *Proc. of the 15th International SPIN Workshop on Model Checking Software (SPIN 2008)*, pages 306–324, 2008.
20. F. Yu, T. Bultan, and O. Ibarra. Relational string verification using multi-track automata. In *Proceedings of the 15th International Conference on Implementation and Application of Automata (CIAA 2010)*, 2010.
21. F. Yu, T. Bultan, and O. H. Ibarra. Symbolic string verification: Combining string analysis and size analysis. In *Proc. of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 322–336, 2009.